Pragma's

product profiles

Issue #33 — The *free* newspaper for Pick™ operating system users. — March 6, 1987

Need to profile your code?

Sooner or later, every programmer is stuck with a project requiring the modification, conversion, or maintenance of a piece of someone else's software. Sometimes the software will be small, well-written, structured, straight-forward, and (in rare cases) documented. More often, the code is huge, messy, unstructured, obscure, and barely understandable.

— Pragma #1

I find that the largest obstacle I have to tuning systems for speed is finding out what the systems are doing. Much of the code I work on is so convoluted that it is difficult to figure out what the code is doing, much less make it faster.

— Brian Gulino, in Product Profiles #32

A valuable measurement tool... is a profiler — a program which counts the number of times each statement in (another) program is executed, by adding counting statements to source statements before compilation, then neatly listing the accumulated information alongside the original source program after the modified program has run. Just knowing the number of times each statement has been executed tells you what parts of the program are most often executed and are thus most likely to dominate the execution time. You can see what parts of the program have never been executed, which may indicate useless code, inadequate testing, or just plain errors. And you can detect performance bugs — regions which are executed more than they should be, such as computations inside loops when they don't have to be.

— Software Tools, by B. Kernighan and P. J. Plauger

Let's construct a profiler for Pick BASIC programs. What we need first of all is a list of counters. If we're profiling a program containing fifty lines of code, then we need fifty counters. Before each line in the program is executed, its corresponding counter is incremented by one. When the program is done executing, the counters will show how many times each line of the program was reached.

An array is a natural data structure for maintaining a list of counters. Let's use a dynamic array named `STMT` (short for "statement"). Our profiler needs to input a source program and then output a modified version of the program, making sure

Pragma's Product Profiles is published periodically by Semaphore Corporation, 207 Granada Drive, Aptos, CA 95003, (408) 688-9200.

Subscriptions are free, but are available only to Pick users with USA mailing addresses.

All material received will be considered for publication. Semaphore reserves the right to edit all submittals.

Semaphore cannot be responsible for lost issues or issues returned to us because of address changes.

Does Your Mailing Label Say RENEW?

If so, your subscription is about to EXPIRE!

To avoid missing your next free issue of *Product Profiles*, renew IMMEDIATELY by filling out and returning the adjacent coupon for a free subscription!

The last two digits of your mailing label number indicate the number of the last issue you'll receive.

Pragma's Product Profiles is completely FREE!

Name _____

Company _____

Address _____

City/State _____ ZIP _____

Telephone _____

I use the Pick operating system and my mailing address is in the USA.
Please continue sending me *Pragma's Product Profiles* free of charge.

Signature _____ Date _____

Return this form to:
Pragma • 207 Granada Drive • Aptos, CA 95003

Pragma's Product Profiles
207 Granada Drive
Aptos, CA 95003

BULK RATE
U.S. POSTAGE
PAID
APTOS, CA 95003
Permit No. 67

Address Correction Requested

each program line increments one of the attributes in STMT. For example, our profiler should turn this program

```
PRINT "WHAT IS YOUR NAME":
INPUT NAME
PRINT "YOUR NAME IS ":NAME
STOP
```

into this program

```
STMT<1>=STMT<1>+1
 PRINT "WHAT IS YOUR NAME":
STMT<2>=STMT<2>+1
 INPUT NAME
STMT<3>=STMT<3>+1
 PRINT "YOUR NAME IS ":NAME
STMT<4>=STMT<4>+1
 STOP
```

When the "profiled" version of the program finishes executing, we can output the counts in STMT to find out how many times each line of the program was reached.

Unfortunately, our profiler has to be a little more clever than just a text editor that prefixes each line with a STMT assignment statement. First of all, how are the counts in STMT actually output? As soon as a profiled program reaches a STOP statement, the program terminates and all the accumulated counts in STMT are lost. Our profiler will have to be smart enough to find every STOP statement in the original program and modify them to somehow output or save the STMT counts.

Another problem occurs the very first time STMT is incremented. Unless STMT is properly initialized to null or zero, a profiled program will cause a "variable has not been assigned a value..." error.

Can you think of other instances when the profiler shouldn't just insert a STMT assignment statement at the beginning of a line of code? Some statements, like SUBROUTINE, can't be prefixed by executable code. Putting counting code in front of statement labels must also be avoided. For example, even though the profiled statement

```
STMT<2>=STMT<2>+1; 50 CRT
```

will compile without complaint, the statement should really be profiled as

```
50 STMT<2>=STMT<2>+1; CRT
```

so that transfers to the label 50 will always be correctly counted in STMT.

CASE is a particularly tricky statement to profile. For example, the following

```
STMT<22>=STMT<22>+1
 BEGIN CASE
STMT<23>=STMT<23>+1
 CASE X=1 ; CRT "ONE"
STMT<24>=STMT<24>+1
 CASE X=2 ; CRT "TWO"
STMT<25>=STMT<25>+1
 CASE X=3 ; CRT "THREE"
STMT<26>=STMT<26>+1
 END CASE
```

will not compile. The profiler cannot insert code between BEGIN CASE and the first CASE clause, and must instead change the code to

```
STMT<22>=STMT<22>+1
 BEGIN CASE
 CASE X=1 ; CRT "ONE"
STMT<24>=STMT<24>+1
 CASE X=2 ; CRT "TWO"
STMT<25>=STMT<25>+1
 CASE X=3 ; CRT "THREE"
STMT<26>=STMT<26>+1
 END CASE
```

The program named PROFILE on page 6 is just such an intelligent profiler. PROFILE inputs a BASIC program, carefully and correctly inserts STMT assignment statements at all the right places, and then saves the new, profiled version of the program. When the profiled program is compiled and executed, it increments a STMT attribute each time a program line is reached, then it saves the STMT array in the master dictionary just before stopping. Printing the STMT counts alongside the original program listing shows how many times each line in the program was reached during program execution.

PROFILE begins execution at line 5, where it asks for the name of the file containing a syntactically correct BASIC program. The name of the program item is input at line 6, and the original unprofiled source code is then read into the OLD.ITEM variable.

OLD.ITEM is scanned attribute by attribute by the loop in lines 9 through 16. Each line of source code is placed in the TEXT.LINE variable at line 10, and the loop then calls the MODIFY.TEXT.LINE subroutine at line 14, which adds the necessary profiling code to TEXT.LINE as needed. The new, profiled line of code is then placed in NEW.ITEM at line 14, and line 15 outputs an asterisk to show PROFILE's progress, just like a compiler. After all lines in OLD.ITEM have been scanned, modified as necessary, and stored in NEW.ITEM, "PROFILE." is prefixed to the original item name and NEW.ITEM is then written out to the source file at line 17. The profiled source code can then be compiled and executed, after which it leaves an item called STMT in the master dictionary.

The MODIFY.TEXT.LINE subroutine begins at line 20, where it calls GET.TOKEN to get the first "token" in the TEXT.LINE variable. A token can be an integer, an alphanumeric symbol, a quoted string, a carriage return (actually an attribute mark), or some other arbitrary punctuation character such as a comma or plus sign. The GET.TOKEN subroutine gathers the token from TEXT.LINE and saves it in the variable TOKEN (except for carriage returns, which are saved as nulls), along with a number from 1 to 5 to indicate what kind of token it is, which is stored in the TOKEN.TYPE variable. The five possible token types are declared symbolically in line 3. As a convenience for the caller, the GET.TOKEN subroutine also sets the flag named COMMENT if the token is an asterisk, exclamation point, or the word REM.

The MODIFY.TEXT.LINE subroutine checks COMMENT in line 21 just after calling GET.TOKEN, and calls the INSERT.CODE subroutine if a comment has indeed been found. The INSERT.CODE subroutine inserts a STMT assignment statement in TEXT.LINE at the front of the comment, and MODIFY.TEXT.LINE can then return, since the rest of

the comment line can be ignored. But if a comment isn't found, then line 22 is reached and the token must be a number (a statement label) or an alphanumeric symbol, which are the only two possibilities at the start of a line of compilable BASIC source code.

If the first token in TEXT.LINE is a number, MODIFY.TEXT.LINE reaches line 30, and INSERT.CODE is called to insert a STMT assignment statement after the numeric statement label. The variable INSERT.POS tells the INSERT.CODE subroutine at what column in TEXT.LINE to insert assignment statements. A column counter is kept in the COLUMN variable, which is initialized at the start of each TEXT.LINE scan at line 20. COLUMN is incremented as the GET.TOKEN subroutine scans TEXT.LINE, and is left pointing to the next unscanned column position after a token has been found. GET.TOKEN also sets START.COL equal to the position before the current token at line 47. When MODIFY.TEXT.LINE calls INSERT.CODE at line 21, INSERT.POS is set equal to START.COL so that the assignment statement is inserted *before* the comment just found. When MODIFY.TEXT.LINE calls INSERT.CODE at line 30, INSERT.POS is set equal to COLUMN so that the assignment statement is inserted *after* the statement label number just found.

If the first token in a TEXT.LINE is not a number, it must be an alphanumeric symbol, so MODIFY.TEXT.LINE reaches line 23, where it simply returns if the token is the start of a PROGRAM, SUBROUTINE, or COMMON statement, since those statements can't be prefixed by new STMT code. Any other token might still be a symbolic (as opposed to numeric) statement label followed by a colon, so MODIFY.TEXT.LINE calls GET.TOKEN one more time at line 27 to look ahead and see if the next token is indeed a colon. If a colon isn't found, this TEXT.LINE doesn't have a symbolic label, so COLUMN is reset to one in line 27 to make sure the upcoming code insertion goes at the beginning of the source statement. If a colon is found, then the label and colon were just scanned, so COLUMN is left alone to cause the STMT statement to be inserted after the colon. In either case, the call to INSERT.CODE occurs at line 28.

If the first token on a line is the word CASE, then the previous line might end with BEGIN CASE, which means a STMT assignment shouldn't be inserted. While TOKEN always contains the current token, OLD.TOKEN is used to remember the previous token, and OLDEST.TOKEN remembers the token before that. TOKEN and OLD.TOKEN are initialized in line 8, and all three variables are shuffled appropriately in line 45 before each new token is collected. If the first token on a line is the word CASE, and OLDEST.TOKEN is also CASE, then OLD.TOKEN must be a carriage return, and the first CASE clause after a BEGIN CASE has been found.

That's what line 26 tests for, causing STMT code insertion to be skipped if necessary.

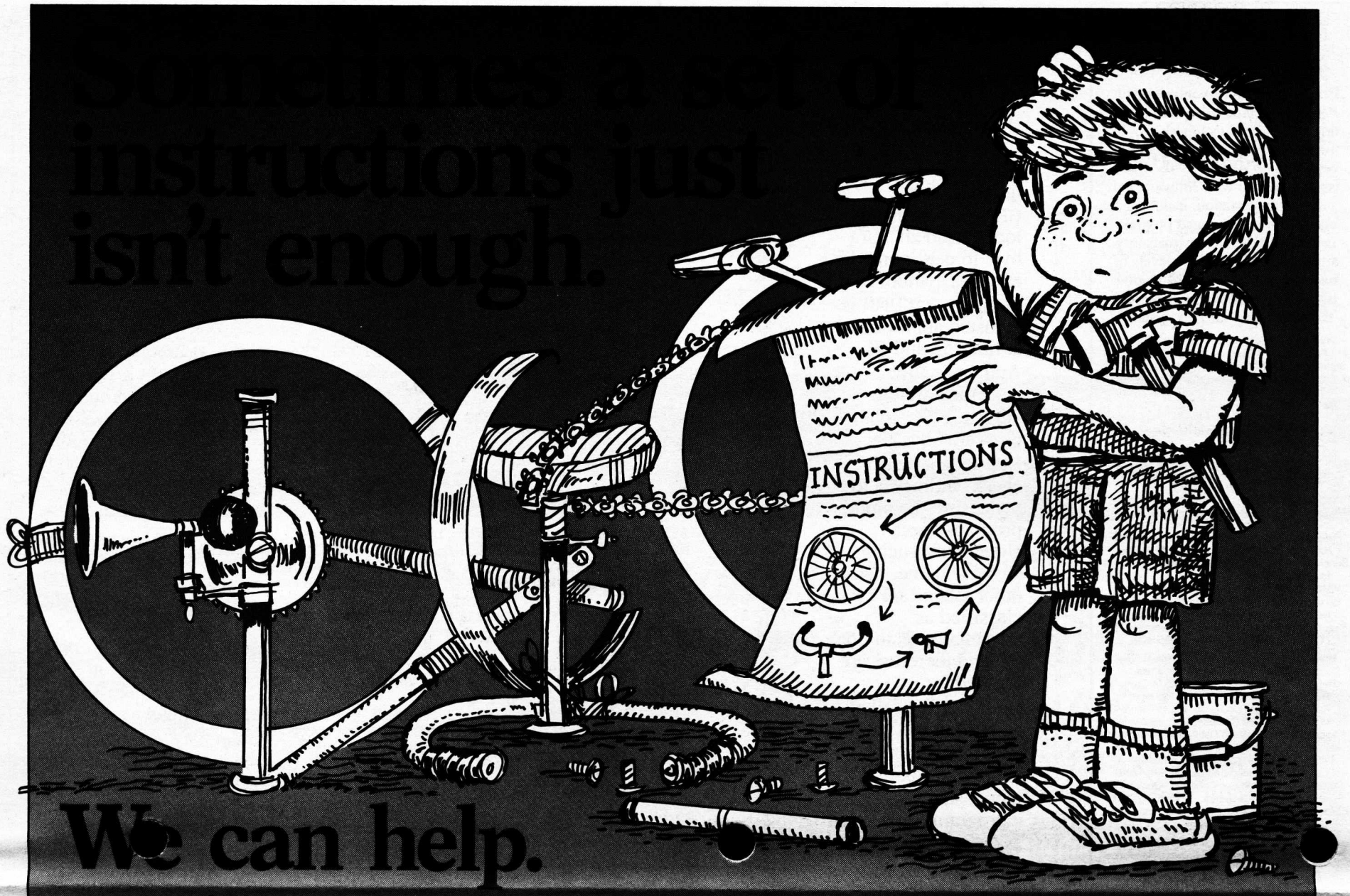
Once code insertion at the beginning of TEXT.LINE has been handled correctly by lines 20 through 30, MODIFY.TEXT.LINE must still scan the rest of the line looking for STOP and ABORT statements, where code is inserted for saving the final STMT counts in the master dictionary. The call to GET.TOKEN in line 32 gets the next token in the statement on the line. If the token starts a comment, TOKEN.TYPE is forced to be a carriage return in line 33 so the LOOP will terminate at line 42, and the rest of the line is ignored. Otherwise, all tokens up to the next semicolon or carriage return are checked. If any token is STOP or ABORT, STMT output code is inserted by lines 36 and 37. When a semicolon is found, the outer loop starting at line 31 is repeated, in case another STOP or ABORT or comment remains. Note that MODIFY.TEXT.LINE always detects and skips comments so that comments containing the words STOP, ABORT, PROGRAM, COMMON, SUBROUTINE, and CASE are ignored, profiling speed is improved, and needless code generation is avoided.

The INSERT.CODE subroutine copies whatever code is stored in NEW.CODE into the TEXT.LINE variable. NEW.CODE contains either text like STMT<3>=STMT<3>+1, which is created by lines 12 and 13, or it contains an OPEN and WRITE statement to save the STMT array in the master dictionary before a STOP or ABORT, which is handled by line 36. Note that OPEN 'MD' ELSE NULL; WRITE... can't be used, since the WRITE wouldn't be executed after a successful open. Also, if the program doesn't contain a STOP or ABORT (relying instead on reaching an END statement to terminate execution), the code to write the STMT array will never be inserted! Since the value of INSERT.POS is controlled by the caller, INSERT.CODE simply copies text on the left of the insertion point into LEFT.PART in line 74, copies text on the right side of the TEXT.LINE into RIGHT.PART in line 75, then rebuilds TEXT.LINE with NEW.CODE in the middle in line 76 and adjusts COLUMN to take the length of the new code into account in line 77.

FIRST.TIME is a flag initialized true in line 7 to indicate that the next call to INSERT.CODE will be the first call ever. When INSERT.CODE detects at line 78 that the very first reference to STMT has just been inserted in TEXT.LINE, line 79 inserts code to initialize STMT to null at the very beginning of TEXT.LINE (which guarantees STMT isn't initialized to null after a statement label), and FIRST.TIME is cleared by line 80 so the initialization code is never re-inserted.

The GET.TOKEN subroutine skips all blanks at line 46 by calling the GET.BYTE subroutine, which simply returns the next character from TEXT.LINE at the current COLUMN position, then increments COLUMN, all at line 72. If the end of a line is reached, GET.TOKEN sets TOKEN.TYPE to 1 at line 48. If the leading BYTE is one of the "letters" equated in line 1 that are allowed to start an alphanumeric symbol, then all subsequent letters and "digits" (equated in line 2) are collected by the loop in lines 51 through 55. Similar collection loops gather quoted

instructions just isn't enough.



We can help.

Diversified Computers presents the first video cassette training series available for the PICK™ operating system. Each course has been carefully tailored to help train you to better use your system.

Training That Really Works

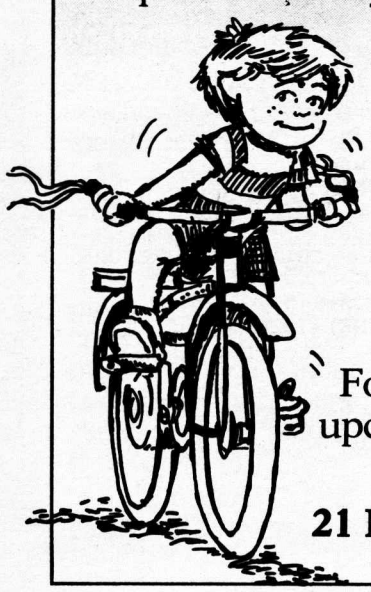
These video training courses are not only the fastest way to learn PICK™, they're also the most thorough and intensive. Specific examples, illustrated step-by-step procedures and hands-on practice sessions guide you to productive and efficient use of your system.

Training You Can Count On Time After Time

These video courses provide you with training that is consistent in quality and content. Now you have training that you can repeat whenever and wherever you need it.

Training Designed Specifically For Organizations Like Yours

Each of the video courses is uniquely designed with the flexibility and ease of use required in today's corporate settings. The videos are structured for use as self-paced individualized instruction, easily accommodating your busy schedule.



The first PICK™ VIDEO SERIES includes five tapes covering the introduction to:

- TCL, EDITOR, PROC
- ACCESS
- DICTIONARIES
- SPOOLER, RUNOFF
- BASIC

DIVERSIFIED COMPUTERS

For more information regarding these and our upcoming video training tapes, write or call today.

21 North Main St. • Suite 203 • Alpharetta (Atlanta), GA 30201 • Phone (404) 475-4601

PICK™ is a registered trademark of PICK Systems

string tokens at line 58 and numeric tokens at lines 61 through 63. Like comments, quoted strings must be skipped by PROFILE so imbedded words like STOP can be ignored. Since GET.TOKEN must look ahead an extra byte to determine the end of a symbol or number, COLUMN is adjusted appropriately at lines 56 and 64. And since PROFILE adds the variable named STMT to all programs it scans, GET.TOKEN also checks at line 68 to make sure the input program doesn't already use such a variable name. All other tokens are classified as miscellaneous in line 65, and therefore are ignored by GET.TOKEN callers.

Executing PROFILE means simply typing in a file name, such as BP, and a program name, such as TEST. PROFILE will then scan TEST, insert various statements, and output the resulting new code with the name PROFILE.TEST. Executing the new PROFILE.TEST program causes the STMT item to be written in the master dictionary when PROFILE.TEST stops. Then a program like

```
OPEN "MD" ELSE STOP
READ STMT FROM "STMT"
  ELSE STOP
CRT "File":
INPUT FILE.NAME
OPEN FILE.NAME
  ELSE STOP
CRT "Item": ; INPUT ID
READ ITEM FROM ID
  ELSE STOP
L = 1
LOOP
  TEXT = ITEM<L>
UNTIL TEXT = "" DO
  PRINT STMT<L> "R#10":
  PRINT " ":TEXT
  L = L+1
REPEAT
STOP
END
```

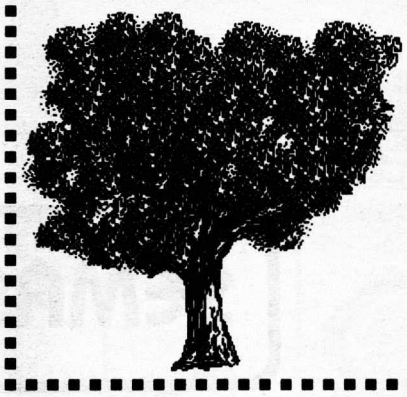
can produce a listing of the original TEST program, with each line prefixed with a count showing the number of times each line was reached, thereby revealing the kinds of information mentioned in the introductory quote from *Software Tools*. For example, using PROFILE to profile itself shows that the GET.BYTE subroutine is invoked 4,022 times while scanning PROFILE's own code. Perhaps GET.BYTE could be inline code instead of a subroutine, to gain some speed? The next most frequently executed statement was line 53, executed 2,211 times. Perhaps the INDEX function should be replaced with a faster way of checking if a byte is a letter?

One disadvantage of profiled code is that it slows down considerably. Executing all those STMT assignment statements takes time, especially for a dynamic array. Fortunately,

```
   PROFILE
001 EQU ltrs TO "$ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
002 EQU digs TO "0123456789."
003 EQU return TO 1, number TO 2, symbol TO 3, string TO 4, misc TO 5
004 *
005 CRT "File": ; INPUT FILE.NAME ; OPEN FILE.NAME ELSE STOP
006 CRT "Item": ; INPUT ITEM.ID ; READ OLD.ITEM FROM ITEM.ID ELSE STOP
007 NEW.ITEM = "" ; LINE.NUMBER = 1 ; FIRST.TIME = 1
008 TOKEN = "" ; OLD.TOKEN = ""
009 LOOP
010   TEXT.LINE = OLD.ITEM<LINE.NUMBER>
011 UNTIL TEXT.LINE = "" DO
012   STMT.LINE.NUM = "STMT<":LINE.NUMBER:">"
013   NEW.CODE = " ":STMT.LINE.NUM:"=":STMT.LINE.NUM:"+1;"
014   GOSUB MODIFY.TEXT.LINE ; NEW.ITEM<LINE.NUMBER> = TEXT.LINE
015   CRT "*": ; LINE.NUMBER = LINE.NUMBER+1
016 REPEAT
017 WRITE NEW.ITEM ON "PROFILE." : ITEM.ID
018 STOP
019 *
020 MODIFY.TEXT.LINE: COLUMN = 1 ; GOSUB GET.TOKEN
021 IF COMMENT THEN INSERT.POS = START.COL ; GOSUB INSERT.CODE ; RETURN
022 IF TOKEN.TYPE # number THEN
023   IF (TOKEN = "PROGRAM") ! (TOKEN = "PROG") THEN RETURN
024   IF (TOKEN = "COMMON") ! (TOKEN = "COM") THEN RETURN
025   IF TOKEN = "SUBROUTINE" THEN RETURN
026   IF (TOKEN # "CASE") ! (OLDEST.TOKEN # "CASE") THEN
027     GOSUB GET.TOKEN ; IF TOKEN # ":" THEN COLUMN = 1
028     INSERT.POS = COLUMN ; GOSUB INSERT.CODE
029     END
030   END ELSE INSERT.POS = COLUMN ; GOSUB INSERT.CODE
031 LOOP
032   GOSUB GET.TOKEN
033   IF COMMENT THEN TOKEN.TYPE = return ELSE
034     LOOP UNTIL (TOKEN.TYPE = return) ! (TOKEN = ";") DO
035       IF (TOKEN = "STOP") ! (TOKEN = "ABORT") THEN
036         NEW.CODE = " OPEN 'MD' THEN WRITE STMT ON 'ST[illegible]"
037         INSERT.POS = START.COL ; GOSUB INSERT.CODE
038         END
039       GOSUB GET.TOKEN
040     REPEAT
041     END
042 UNTIL TOKEN.TYPE = return DO REPEAT
043 RETURN
044 *
045 GET.TOKEN: OLDEST.TOKEN = OLD.TOKEN ; OLD.TOKEN = TOKEN
046 LOOP GOSUB GET.BYTE WHILE BYTE = " " DO REPEAT
047 TOKEN = BYTE ; START.COL = COLUMN-1
048 IF TOKEN = "" THEN TOKEN.TYPE = return ELSE
049   BEGIN CASE
050   CASE INDEX(ltrs, BYTE, 1)
051     LOOP
052       GOSUB GET.BYTE
053     WHILE (INDEX(ltrs,BYTE,1) ! INDEX(digs,BYTE,1)) & (BYTE#"") DO
054       TOKEN = TOKEN : BYTE
055     REPEAT
056     COLUMN = COLUMN-1 ; TOKEN.TYPE = symbol
057   CASE (BYTE = "'") ! (BYTE = '"') ! (BYTE = '\')
058     LOOP GOSUB GET.BYTE UNTIL (BYTE=TOKEN) ! (BYTE="") DO REPEAT
059     TOKEN.TYPE = string
060   CASE BYTE MATCHES "1N"
061     LOOP GOSUB GET.BYTE WHILE BYTE MATCHES "1N" DO
062       TOKEN = TOKEN : BYTE
063     REPEAT
064     COLUMN = COLUMN-1 ; TOKEN.TYPE = number
065   CASE 1 ; TOKEN.TYPE = misc
066   END CASE
067   END
068 IF TOKEN = "STMT" THEN STOP "STMT already in use!"
069 COMMENT = (TOKEN = "*") ! (TOKEN = "!") ! (TOKEN = "REM")
070 RETURN
071 *
072 GET.BYTE: BYTE = TEXT.LINE[COLUMN,1] ; COLUMN = COLUMN+1 ; RETURN
073 *
074 INSERT.CODE: LEFT.PART = TEXT.LINE[1,INSERT.POS-1]
075 RIGHT.PART = TEXT.LINE[INSERT.POS,LEN(TEXT.LINE)-INSERT.POS+1]
076 TEXT.LINE = LEFT.PART : NEW.CODE : RIGHT.PART
077 COLUMN = COLUMN + LEN(NEW.CODE)
078 IF FIRST.TIME THEN
079   INIT = "STMT='';" ; TEXT.LINE = INIT : TEXT.LINE
080   COLUMN = COLUMN + LEN(INIT) ; FIRST.TIME = 0
081   END
082 RETURN
083 *
084 END
```

profiled programs are usually just run once to gather the STMT statistics. But if you need to speed up the programs being profiled, one easy yet drastic improvement would be to use MATWRITE instead of WRITE in line 36, and then use code like STMT(3) instead of STMT<3> at line 12, which would require changing the STMT=" initialization in line 79 to DIM STMT(x); MAT STMT=", where x is the number of lines in OLD.ITEM, counted just after the READ in line 6.

COMING SOON:
P.U.P.
the Pick Users Pool!



Semaphore welcomes it's newest B-TREE-P customers:

Livermore Police Department
Creative Computer Services
Berelson Company
Minnesota Trade Office

FREE Back Issues!

A few back issues of *Pragma's Product Profiles* are still available while supplies last. To receive your FREE copies, indicate which issues you need and send a stamped, self-addressed envelope to:

Pragma • 207 Granada Dr. • Aptos CA 95003

(Allow 1.5 oz. per issue to compute postage.)

Issue #10: Beyond The Power Of Pick
Issue #11: Open Architecture Status Report
Issue # Impressions Of The Zebra 750
Issue # The Wonders Of An Upgrade
Issue #17: A READ Sometimes Fails
Issue #19: Pick Book A Disappointment
Issue #2 Programming For Speed
Issue # How Files Grow
Issue #23: GA About To Release 3.2
Issue #24: Beyond Pick, Revisited
Issue #26: How To Find Wasted Space
Issue #27: The Myth Of Separation=1
Issue #28: Is BASIC Faster Than Access?
Issue #29: How To Crunch Code
Issue #30: Semaphore Interviews Itself
Issue #32: This Month's Mailbag

*** FOR SALE ***

Microdata Sequel 9000
2 MB Memory
256 MB Disc
40 Ports
1600/3200 Tape Drive
Power Conditioner
3 Prism IV Terminals
2 Cabinets

$55,000.00 O.B.O.
"As is, where is"
Under continuous Microdata service

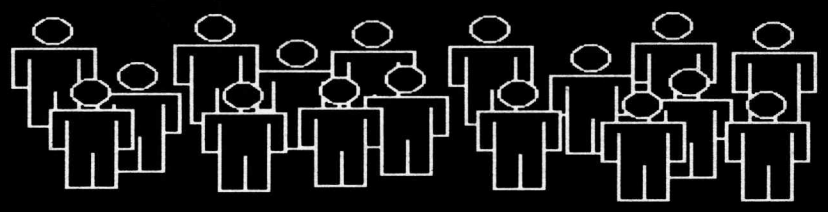
29 Visual 50 Terminals
Available at $150.00 each

(818) 349-5650 Ask for Faye or Ralph

* System available April 1st or sooner

Now available for the first time!

Our Pick™ Mailing List
CREAM of the CROP



Our 3,000 most responsive
Pick users' names and addresses.
Only $150 plus tax for one-up adhesive labels,
sorted, printed, and delivered to you.

Send no payment without first obtaining our order form and one-time use agreement.

P/Mail by Marianne
1893 Nadina Street • Seaside CA 93955

ZEBRA FOR SALE

General Automation Zebra 752

128KB memory (expandable to 640KB)
20MB disk (expandable to 120MB)
6 serial ports (expandable to 18) • 1 parallel port
5 available option slots
5MB backup cartridge drive (expandable to 1/4" tape)
12 backup cartridges • All manuals

With Pick O/S, Jet, AccuPlot,
CompuSheet, and B-TREE-P.
This is an extremely reliable, lightly used machine.

$2,900

Semaphore Corporation
207 Granada Dr. • Aptos CA 95003 • (408) 688-9200

SEMAPHORE

SEMAPHORE CORPORATION • 207 Granada Drive • Aptos, California 95003 • (408) 688-9200

Dear Pick™ user,

These fifty companies* use a Pick computer just like you do, but their computer can do some special things that your computer can't:

Long Beach Community Services • Halprin Supply Co. • Casualty Underwriters Inc. • University of California • Distributed Logic Corp. • Jet Electronics & Technology Inc. • National Center for Atmospheric Research • Trudell Trailer Sales Inc. • Stewart Co. • Computyme • Data Operating Systems Inc. • Tel-A-Train Inc. • Information Technology Consultants • Life & Health Insurance Co. of America • Flynt Systems Corp. • System Works Inc. • Miami Trading Enterprises • Generation Research • Multisystems Inc. • Infocel • Cooke Data Systems Inc. • John Klein & Assoc. Inc. • Condominium Insurance • Penn Independent Assoc. Inc. • Mark Card • Chandler Lumber Co. • Eye Care & Surgery Center • Chicago Kenworth • Wofford College • Conston Inc. • Assertive Systems • Office Works • Cornell University • City of Irvine • Excalibur Computer Systems Inc. • ADDS Inc. • Specs Music • Specialty Underwriters Inc. • Shoob Photography • May Trucking • Sierra Software Inc. • Medical Accounting Systems • AIPAC • NORPAC • Martin Cadillac Co. Inc. • Capital Software Ltd. • Oman Publishing • Reinsurance Assoc. • Hubert Distributing Co.

Their Pick computers can scroll through files, forward *or* backward, an item at a time or a page at a time, in *any* sort order. They don't have to wait for SORTs or SELECTs. They can immediately find any record in any file just by typing one or more starting characters that match *any* field in the record.

Why are these companies special? Because they purchased B-TREE-P, software for using B-trees on Pick computers. B-trees allow any of the data in any of your Pick files to be instantly located, displayed, and processed in any sort order, without having to wait for SORT or SELECT commands.

B-TREE-P and a few minor modifications to your existing data entry programs are all that is necessary for you to immediately be able to search, display, and browse through your data quickly and conveniently.
Modifications to your existing data files are absolutely unnecessary!
B-trees do *not* use inverted files, cross-reference tables, or other similar inefficient indexing schemes.

Why not join the ranks of B-TREE-P users who can instantly locate and display their data any way they want, without having to wait for endless SORTs and SELECTs? To order, just send your name, address, telephone number, and your check for $395 payable to Semaphore Corporation to the address on our letterhead. We'll send you complete B-TREE-P source code listings and all necessary documentation.

* This is not an endorsement by any of the companies listed.